Executable code generation from CSP specifications

Federico Rossi

Programming Language Analysis and Formal construction of programs, Bachelor Degree in Computer Science, National University of Rosario, Argentina
Email: frossi.933@gmail.com

Communicating Sequential Processes (CSP) is a formal language for describing patterns of interaction in concurrent systems. CSP has been practically applied in industry for specifying and verifying the concurrent aspects of a variety of different systems. But those applications are still in a different level of abstraction than the source code and we can identify a gap between the specifications and the implementation of a system. This project aims to fill that gap and increase the practical approach, giving a tool for deriving executable code from a formal specification and some functions written in Haskell. First of all, a domain specific language for a slightly different version of CSP, which is called CSPi, is defined. Then the evaluator is developed using Haskell, based on an operational semantics. On the other hand, this development includes formal proofs within the Coq system that ensure the correctness of the evaluator.

Keywords: Communicating Sequential Processes, Coq system, Formal methods, ...

1. INTRODUCTION

CSP was first described in a 1978 paper by Tony Hoare (1). It has been applied in industry developments, such as the T9000 Transputer,[7] as well as a secure ecommerce system.[8] The theory of CSP itself is also still the subject of active research. Industrial application to software design has usually focused on dependable and safety-critical systems. For example, the Bremen Institute for Safe Systems and Daimler-Benz Aerospace modeled a fault management system and avionics interface (consisting of some 23,000 lines of code) intended for use on the International Space Station in CSP, and analyzed the model to confirm that their design was free of deadlock and livelock.[13][14] The modeling and analysis process was able to uncover a number of errors that would have been difficult to detect using testing alone. All these examples show us the importance of specifying our systems using formal languages to produce high quality software. But sometimes in industry, the specification stage is considered not important because it doesn't take part directly in the source code or executable program. The aim of this coursework is to engage both stages in the development and create a strong dependency between them, ensuring that the program obtained respects its formal specification. CSPi is a simplified version of CSP with new notions about events. Each event can be associated with a function implemented in Haskell. These functions may be a predicate (which returns a boolean) or an action (which performs an I/O activity). The main idea is to describe and model the logic of the system using CSPi and implement the interface with the outer world using Haskell. Let's think a small system where we can apply these ideas. There is a cell production with two robotic arms, two conveyor belts and one press. Each arm must take one item from one of the belts and drop it on the press (each arm works on one belt). Each conveyor belt triggers a signal when one item is coming. The press receives signals to press and remove resultant item, and triggers a signal when it's done processing and it is free to work again. In this case, we can model the logic of the system using CSPi, this is independent of the specific hardware. Then, we have to implement the interface in Haskell, that is, make the predicates that tell us when the signals are triggered and the functions that control the arms, belts and press. Again, those functions are not related with the system's model. The independence is always a desired outcome. It helps minimize the cost of a future change. The fact of modeling the system in a formal language, such as CSP, gives us many adventages. It allows us to describe the system in a higher level of abstraction. Also, one can formally demonstrate properties of the program, like in the examples above. I introduce the CSPi language in the next section, then in Section 2 I analyse the Haskell implementation of the evaluator and finally in Section 3 I describe and show the Coq formalization and proofs carried out.

2. RELATED WORK

This project differs from CSP tools such as FDR (029), that are designed to formal reasoning, e. g. proving that one system is a refinement of another. The work described here is much less complete in the formal sense; FDR will cater for all possible traces of a CSP system, the code generated by the compiler described here executes an arbitrary trace of the system. That trace may be different each time the program is executed, or the same, consistent with the expected rules of non-determinism. The advantage of this work is that it allows relatively large CSP systems to be exercised, since state-space explosion is not an issue.

There are extensions or libraries for many general purpose programming language providing concurrent models based on Hoare's CSP, such as JCSP, C++CSP, CTC++, .NET CSP, etc. Each of those implementations requires a background knowledge about the programming language itself. Furthermore, these extensions are also limited by the characteristics of the base language. These are useful tools for developing concurrent systems programming in a general purpose language, but I have talked about the big importance of writing a system specification, so given that, the main drawback of this approach is the necessity of translating the formal specification in the chosen programming language. Instead, CSPi connects directly a system specification with its implementation, avoiding programmer's subjective interpretations.

Finally, the most related work is Ocamm programming language. Occam is a parallel programming language developed by David May [May, 83] at Inmos Limited, Bristol, England. Using CSP as a basis, the researchers at Inmos developed an Occam concurrency model. From the Occam model, they developed the programming language. It is the native language for their transputer microprocessors, but implementations for other platforms are available nowadays.

It worth to say that all works mentioned previously provide a more complete set of features. But this project aims to carry out a different approach to generate executable code from a specification.

3. CSPI LANGUAGE

CSPi contains a subset of CSP operands and functionalities.

$$Proc : STOP \tag{1}$$
$$|\, SKIP \tag{2}$$
$$|\, '('Proc')' \tag{3}$$
$$|\, Event' - >' Proc \tag{4}$$
$$|\, RefProc \tag{5}$$
$$|\, Proc'||'Proc \tag{6}$$
$$|\, Proc'[]'Proc \tag{7}$$
$$|\, Proc'/|'Proc \tag{8}$$
$$|\, Proc';' Proc \tag{9}$$
$$|\, Proc'| >' Proc \tag{10}$$
$$\tag{11}$$

4. FORMAL PROOFS

The objective of this section is to introduce the project formalization within the Coq System. It attemps to prove that executable programs generated behave like their specifications. First of all, I present the event representation, then I describe an operational semantic for CSPi, based on (82). After that, I define what I'll consider a trace in my system. Finally, I show that the two main evaluation functions respect the semantic and return a valid trace.

4.1. Events Representation

In previous sections I informally described the notions and meanings of the different types of events. Now it's time to define them in a formal way. In addition, I consider a new kind of event, it is called Epsilon and performs a silent transition. It can be seen as an event to "rewrite" or "redefine" some process. It is going to be more clear in the eval function definition. It's important to remark that Epsilon is transparent to the user, that is, it doesn't interact with the outer world. Some previous definitions are required:

```
Definition IdEv : Set := nat.
Definition Input : Set := bool.
Parameter Output : Set.
Definition Pred : Set := nat -> Input.
Definition Act : Set := nat -> Output.
```

I need to identify each event, for that purpose I use IdEv. Input and Output symbolize a return type of some function after carrying out an I/O activity. Input must be a boolean but Output can be any type. Finally, Pred and Act represent the functions associated with the events. In this case, they receive a natural argument but it is just a simplification. Thus, the events have the following definition:

```
Inductive Event : Set :=
  Eps : Event
  EIn : IdEv -> Pred -> Event
  EOut : IdEv -> Act -> Event.
```

Afterwards, I define an equality between events and it is based on comparing only id's. Two events are equal if and only if their id's are equal or both are Epsilon.

```
Inductive Beq_event : Event -> Event -> Prop :=
| beq_eps : Beq_event Eps Eps
| beq_inin : forall (i:IdEv)(p1 p2:Pred),
    Beq_event (EIn i p1)(EIn i p2)
| beq_inout : forall (i:IdEv)(p:Pred)(a:Act),
    Beq_event (EIn i p)(EOut i a)
| beq_outout : forall (i:IdEv)(a1 a2:Act),
    Beq_event (EOut i a1)(EOut i a2)
| beq_outin : forall (i:IdEv)(a:Act)(p:Pred),
    Beq_event (EOut i a)(EIn i p).
```

At last, I create a simple structure to specify a set of events. It is just a list with no repetition and the basics operations of set, such as member, add, intersection, union and difference. This structure is used when the eval function calculates, which I called, the menu of a process, that is, the set of possible events ready to evolve. In the future, it would be interesting to add support to channels, including them as a new kind of event.

4.2. Semantics

As a first step I give an operational semantics following the main ideas expressed in (92) by Manfred Broy. An operational Semantics is recommended for several reasons. First, it does not lead into the complex problems of power domains that have to be dealt with in mathematical semantics. Second, formalized operational semantics guarantees that the concepts incorporated in the language actually can be implemented. Third, it serves as a complementary definition such that the "correctness" of mathematical ("denotational") semantics can be verified. Fourth, it helps to understand where the particular problems of mathematical semantics come from, since the "abstraction" mapping the operational semantics into the mathematical one can be made explicit. The semantics is given by the introduction of a ternary relation:

```
  Op: Proc -> Event -> Proc -> Prop
```

For better readability I write:
$P \xrightarrow{e} QforOpPeQ$

This means that the process P may evolve through the event e (maybe performing the action associated with e) and then behave like the process Q. Thus, the definition in Coq is shown in figure 1 and the respective rules are shown in figure 2. Some aspects to remark, first in the parallel operator is needed to specify a set of events, which must be the events in common between both processes parallelized. If there is an common event not included in that set then the evaluator does not guarantee its correct evaluation. Second ...

4.3. Small step evaluation

Now, I can define the function which is going to make a small step in the evaluation procedure. It is called smallstep_eval and has the following prototype:

```
Fixpoint smallstep_eval (p:Proc) (e:Event)
                : option Proc
```

It is a recursive function that takes a process and an event and returns possibly a new process. It may fails, so its return type is an option. The complete function definition can be seen in figure 3. After that, my primary objective is to prove the correctness of it and that is achived constructing a proof for the theorem below:

```
Theorem sseval_correct:
  forall (e:Event)(p q:Proc),
    smallstep_eval p e = Some q -> Op p e q.
```

It states that if small step function is apply to some process and event and it returns a new process, then that evolution is part of the semantics. It is proven by induction on the process p and making some case analysis. The proof is too long to be included in this document, it is available along with complementary code in the files "eval.v" and "event.v".

4.4. Big step evaluation

Now it's time to think about the entire process execution. The concept of trace is used to model the resultant behaviours of process execution. Its definition is below:

```
CoInductive Trace : Set :=
  nilt : Trace
| const : Output -> Trace -> Trace.
```

Now I have a coinductive type because It's necessary to represent possible infinite runnings, such as the cell production system described before. But It's also possible to have a finite trace, for that reason there is a nil constructor. A finite trace is always obtained when the process reduces to skip or stop. When it's the case of skip the system finished correctly and when it's the case of stop that symbolizes an erroneous outcome. Some new parameters included are:

```
Parameter empty : Output.
Parameter tick : Output.
Parameter execAct : Event -> Output.
Parameter test : Pred -> bool.
Parameter rand : unit -> bool.
```

empty and tick are special outputs that indicate no action performs and an appropriate finish, respectively. execAct serves as the representation of execute an action associated with an event. If there is no action or it is applied to ϵ, it returns empty. test function gives the current state of a predicate. Lastly, rand illustrates a function that returns a bool randomly. Thereafter, I define what I will consider an acceptable trace for a specific process. This is the isTrace relation presented below in figure 4. Finally, I introduce the main function eval, which formalizes the concrete outcome of this project. Its definition is shown in figure 5. Eval function receives a process and returns a trace which represents the execution of the process. In the general case (the process is not skip or stop), it calculates the menu of the process and chooses randomly an event to perform a small-step evaluation. If that results None the evaluation failed and nil trace is returned. If that results some new process, it execute the action associated with the event chosen and continues recursively.

4.5. Divergence

The diverge predicate is introduced to eliminate those process which can produce an infinite loop of non-actions, that is, they may evolve only with the epsilon event. Formally, If there is a sequence of processes Pii N with Po =p1 and Op Pi Eps Pi+1 then I say that p1 has the possibility to diverge. The figure 6 shows its definition in Coq. The idea is to avoid infinite application of rewriting rules, such as P := id "P"

4.6. Main theorem

It's time to prove the principal theorem which states that the evaluator works correctly according to the semantics defined. The theorem statement is as follows:

```
Theorem eval_correct: forall (p:Proc),
       ~(Diverge p) -> isTrace (eval p) p.
```

I need to prove that eval function returns one possible trace of the process received and it just requires a non-diverging process as argument. The complete proof is shown in "eval.v" file. It uses the cofix tactic to construct a co-recursive demonstration and then performs successive cases analysis

FIGURE 1. Operational Semantics of CSPi

```
Inductive Op : Proc -> Event -> Proc -> Prop :=
| prefok:   forall (e0 e:Event)(p:Proc), Beq_event e0 e -> Op (pref e0 p) e p
| choil:    forall (e:Event)(p q r:Proc), Op p e q -> e <> Eps -> Op (choi p r) e q
| choir:    forall (e:Event)(p q r:Proc), Op p e q -> e <> Eps -> Op (choi r p) e q
| choiepsl: forall (p q r:Proc), Op p Eps q -> Op (choi p r) Eps (choi q r)
| choiepsr: forall (p q r:Proc), Op p Eps q -> Op (choi r p) Eps (choi r q)
| choistop: Op (choi stop stop) Eps stop
| skipl:    forall (p:Proc), Op (choi skip p) Eps p
| skipr:    forall (p:Proc), Op (choi p skip) Eps p
| seql:     forall (e:Event)(p q r:Proc), Op p e q -> Op (seq p r) e (seq q r)
| seqskip:  forall (p:Proc), Op (seq skip p) Eps p
| seqstop:  forall (p:Proc), Op (seq stop p) Eps stop
| sync:     forall (e:Event)(p q r t:Proc)(s:EvSet), Op p e q -> Op r e t -> isMember e s -> Op (par s p
| parl:     forall (e:Event)(p q r:Proc)(s:EvSet), Op p e q -> ~(isMember e s) -> Op (par s p r) e (par s
| parr:     forall (e:Event)(p q r:Proc)(s:EvSet), Op p e q -> ~(isMember e s) -> Op (par s r p) e (par s
| parskip:  forall (s:EvSet), Op (par s skip skip) Eps skip
| idref:    forall (n:Name), Op (id n) Eps (Env n).
```

FIGURE 2. Semantics Rules of CSPi

$$\frac{}{(e \to P) \overset{e}{\to} P}\ PrefOk \qquad \frac{(P \overset{e}{\to} Q) \quad e \neq \epsilon}{P \Box R \overset{e}{\to} Q}\ ChoiL \qquad \frac{(P \overset{e}{\to} Q) \quad e \neq \epsilon}{R \Box P \overset{e}{\to} Q}\ ChoiR$$

$$\frac{(P \overset{\epsilon}{\to} Q)}{P \Box R \overset{\epsilon}{\to} Q \Box R}\ ChoiEpsL \quad \frac{(P \overset{\epsilon}{\to} Q)}{R \Box P \overset{\epsilon}{\to} R \Box Q}\ ChoiEpsR \qquad \frac{}{Stop \Box Stop \overset{\epsilon}{\to} Stop}\ ChoiStop$$

$$\frac{}{Skip \Box P \overset{\epsilon}{\to} P}\ SkipL \qquad \frac{}{P \Box Skip \overset{\epsilon}{\to} P}\ SkipR \qquad \frac{P \overset{e}{\to} Q}{P; R \overset{e}{\to} Q; R}\ SeqL$$

$$\frac{}{Skip; P \overset{\epsilon}{\to} P}\ SeqSkip \qquad \frac{}{Stop; P \overset{\epsilon}{\to} Stop}\ SeqStop \quad \frac{P \overset{e}{\to} Q \quad R \overset{e}{\to} T \quad e \in \mathcal{S}}{P \parallel_{\mathcal{S}} R \overset{e}{\to} Q \parallel_{\mathcal{S}} T}\ Sync$$

$$\frac{P \overset{e}{\to} Q \quad e \notin \mathcal{S}}{P \parallel_{\mathcal{S}} R \overset{e}{\to} Q \parallel_{\mathcal{S}} R}\ ParL \quad \frac{P \overset{e}{\to} Q \quad e \notin \mathcal{S}}{R \parallel_{\mathcal{S}} P \overset{e}{\to} R \parallel_{\mathcal{S}} Q}\ ParR \quad \frac{}{Skip \parallel_{\mathcal{S}} Skip \overset{\epsilon}{\to} Skip}\ ParSkip$$

$$\frac{}{IdN \overset{\epsilon}{\to} EnvN}\ IdRef$$

FIGURE 3. smallstep_eval definition

```
Fixpoint smallstep_eval (p:Proc) (e:Event) : option Proc :=
  match p, e with
  | stop, v => None
  | skip, v => None
  | pref v r, v' => if beq_event v v' then Some r else None
  | choi stop stop, Eps => Some stop
  | choi skip p2, Eps => Some p2
  | choi p1 skip, Eps => Some p1
  | choi p1 p2, Eps => match smallstep_eval p1 Eps, smallstep_eval p2 Eps with
                         None, None => None
                       | Some p3, None => Some (choi p3 p2)
                       | None, Some p4 => Some (choi p1 p4)
                       | Some p3, Some p4 => Some (choi p3 p2) (* TODO: random *)
                       end
  | choi p1 p2, v => match smallstep_eval p1 v, smallstep_eval p2 v with
                       None, None => None
                     | Some p3, None => Some p3
                     | None, Some p4 => Some p4
                     | Some p3, Some p4 => Some p3 (* TODO: random *)
                     end
  | seq skip p, Eps => Some p
  | seq stop p, Eps => Some stop   (* ?? *)
  | seq p1 p2, v => match smallstep_eval p1 v with
                      Some p3 => Some (seq p3 p2)
                    | None => None
                    end
  | par s skip skip, Eps => Some skip
  | par s p1 p2, v => if isElem v s then match smallstep_eval p1 v, smallstep_eval p2 v with
                                           Some p3, Some p4 => Some (par s p3 p4)
                                         | _ , _ => None
                                         end
                                    else match smallstep_eval p1 v, smallstep_eval p2 v with
                                           Some p3, None => Some (par s p3 p2)
                                         | None, Some p4 => Some (par s p1 p4)
                                         | _, _ => None
                                         end
  | id n, Eps => Some (Env n) (* ?? *)
  | p , v => None
  end.
```

FIGURE 4. isTrace predicate

```
CoInductive isTrace : Trace -> Proc -> Prop :=
  skip_tick : isTrace (tick :: nilt) skip
| empt:       forall (t:Trace)(p1 p2:Proc), Op p1 Eps p2 ->
                                    isTrace t p2 ->
                                    isTrace (empty :: t) p1
| nil_is_t: forall p:Proc, isTrace nilt p
| preft :   forall (t:Trace)(p:Proc)(e1 e2:Event), Beq_event e2 e1 -> isTrace t p
                                                               -> isTrace ((execAct e1)::t) (pref e
| choitl :  forall (t:Trace)(p1 p2:Proc), isTrace t p1 -> isTrace t (choi p1 p2)
| choitr :  forall (t:Trace)(p1 p2:Proc), isTrace t p2 -> isTrace t (choi p1 p2)
| seqt:     forall (t:Trace)(p1 p2 p3:Proc)(e:Event), isTrace t (seq p1 p2) -> Op p3 e p1
                                                                    -> isTrace ((execAct e) :: t)
| parsynct: forall (t:Trace)(e:Event)(s:EvSet)(p1 p2 p3 p4:Proc),
                 isMember e s -> Op p3 e p1 -> Op p4 e p2 -> isTrace t (par s p1 p2)
              -> isTrace (execAct e :: t) (par s p3 p4)
| parlt:    forall (t:Trace)(e:Event)(s:EvSet)(p1 p2 p3:Proc), ~isMember e s
                                                                                -> Op p3 e p1
                                                                                -> isTrace t (
                                                                                -> isTrace (ex
| parrt:    forall (t:Trace)(e:Event)(s:EvSet)(p1 p2 p3:Proc), ~isMember e s
                                                                                -> Op p3 e p2
                                                                                -> isTrace t (
                                                                                -> isTrace (ex
```

FIGURE 5. eval definition

```
CoFixpoint eval (p:Proc) : Trace := match p with
  stop => nilt
| skip => tick :: nilt
| q => match choose (menu q) with
         None => nilt
       | Some e => match smallstep_eval q e with
                     None => nilt
                   | Some r => execAct e :: (eval r)
                   end
       end
end.
```

FIGURE 6. Diverge definition

```
CoInductive Diverge : Proc -> Prop :=
  lockdiv: forall n:Name, Env n = (id n) -> Diverge (id n)
| loopdiv: forall n:Name, Diverge (Env n) -> Diverge (id n)
| prefdiv: forall (p:Proc)(e:Event), Diverge p -> Diverge (pref e p) (* ??*)
| choildiv: forall p q:Proc, Diverge p -> Diverge (choi p q)
| choirdiv: forall p q:Proc, Diverge p -> Diverge (choi q p)
| seqldiv: forall p q:Proc, Diverge p -> Diverge (seq p q)
| seqdiv: forall p q:Proc, Diverge q -> Diverge (seq p q)
| parldiv: forall (p q:Proc)(s:EvSet), Diverge p -> Diverge (par s p q)
| parrdiv: forall (p q:Proc)(s:EvSet), Diverge p -> Diverge (par s q p).
```